Instruction-level Characterization of Scientific Computing Applications Using Hardware Performance Counters

Yong Luo Kirk W. Cameron
Scientific Computing Group
Mail Stop B256, CIC-19
Los Alamos National Laboratory
Los Alamos, NM 87545
{yongl, kirk}@lanl.gov

Abstract

Recently, advanced microprocessors have incorporated hardware performance counters in their design allowing for new types of analysis via empirical methods. The goal of this analysis continues to be the discovery of analytical/empirical methods to evaluate performance of scaling codes on today's advanced CPU's and to predict effects of architectural advances on current applications. In this paper, we provide an instruction-level characterization derived empirically in an effort to demonstrate how architectural limitations in underlying hardware will affect the performance of existing codes. In particular, we focus on scientific applications of interest to the DOE ASCI (Accelerated Strategic Computing Initiative) community. Preliminary results provide promise in code characterization, and empirical/analytical modeling. These include the ability to quantify outstanding miss utilization and stall time attributable to architectural limitations in the CPU and the memory hierarchy. This work further promises insight into quantifying bounds for CPI_0 or the ideal CPI with infinite L1 cache. In general, if we can characterize workloads using parameters that are independent of architecture, such as this work, then we can more appropriately compare different architectures in an effort to direct processor/code development.

Introduction

Workload characterization has been proven an essential tool to architecture design and performance evaluation in both scientific and commercial computing areas. Traditional workload characterization techniques include FLOPS rate, cache miss ratios, *CPI* (cycles per instruction or *IPC*, instructions per cycle) etc. With the complexity of sophisticated modern superscalar microprocessors, these traditional characterization techniques are not powerful enough to pinpoint the performance bottleneck of an application on a specific microprocessor. They are also incapable of immediately demonstrating the potential performance benefit of any architectural or functional improvement in a new processor design. To solve these problems, many people rely on simulators, which have substantial constraints especially on large-scale scientific computing applications. This paper presents a new technique of characterizing applications at the instruction level using hardware performance counters. It has the advantage of collecting instruction-level characteristics in a few runs virtually without overhead or slowdown. A variety of instruction counts can be utilized to calculate some average abstract workload parameters corresponding to microprocessor pipelines or functional units. Based on the microprocessor architectural constraints and these calculated abstract parameters, the architectural performance bottleneck for a specific application can be estimated. In particular, the analyzed results can provide some insight to the problem that only a small percentage of processor peak performance can be achieved even for many cache-friendly codes. Meanwhile, the bottleneck estimation can provide suggestions about viable architectural/functional improvement for certain workloads. Eventually, these abstract parameters can lead to the creation of an analytical microprocessor pipeline model and memory hierarchy model.

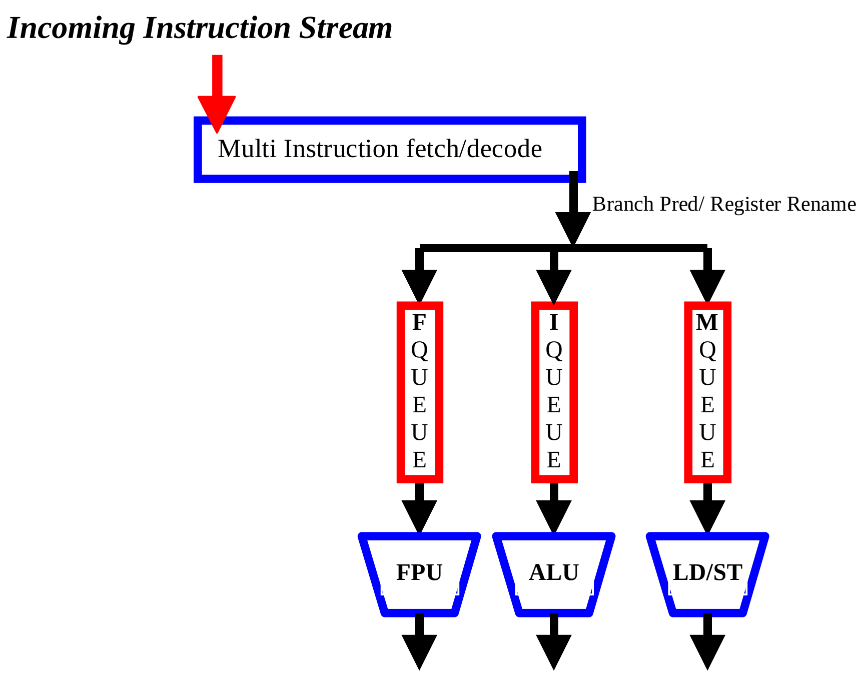
Incoming Instruction Stream
Multi Instruction fetch/decode
Branch Pred/ Register Rename
F QUEUE
I QUEUE
M QUEUE
FPU
ALU
LD/ST

Figure 1 General pipeline model for CPU only

This paper describes the application of this technique on two SGI R10000-based systems: Origin2000 and PowerChallenge, using the SGI performance counter tool *perfex*. Some results are directly validated by the empirical memory model [1] and the statistic model [5].

Methodology

The original motivation of creating an analytical pipeline model for a superscalar microprocessor leads to the definition of a set of workload parameters. We focus on the importance of using instruction-level parameters to characterize a workload so as to associate the workload performance behavior with the microprocessor architecture. Common architectural features of many modern superscalar microprocessors can be generalized as separated pipelines for functional units (Figure 1): one or more integer pipeline(s) for ALU(s), one or more floating-point pipeline(s) for FPU(s), and one or more memory operation pipeline(s) for load/store unit(s). We incorporate memory hierarchy influence on stalls within the microprocessor in the general diagram provided in Figure 2.

A branch prediction unit is usually employed before multiple decoded instructions are dispatched to these pipelines. In applications with a small percentage of branches and high branch prediction hit ratio, the execution of these applications can be virtually viewed as feeding integer, floating-point, and memory instructions into three pipeline queues. At the end of each queue, functional units execute the instructions at a preset rate, e.g. two floating-point instructions per cycle, one memory instruction per cycle, etc. The out-of-order execution feature provides the ability of resolving data dependency within or between these pipeline queues. Therefore, the distributions or the mixture pattern of these three types of instructions in the application instruction flow shall essentially determine the instruction execution rate (*IPC*), ignoring the memory hierarchy effect. This is the well-known IPC_0 or CPI_0, which represents the effective application performance on a specific microprocessor without memory slowdown.

Efficiently measuring these instruction-level workload parameters is the key component to code characterization. The large scale of our application workloads and the big slowdown of detailed instruction-level simulations determine that the measurement of the workload characteristics must resort to some methods other than simulators. On-chip hardware performance counters widely available on recent generation microprocessors become good candidates for extracting these functional-unit-related parameters.

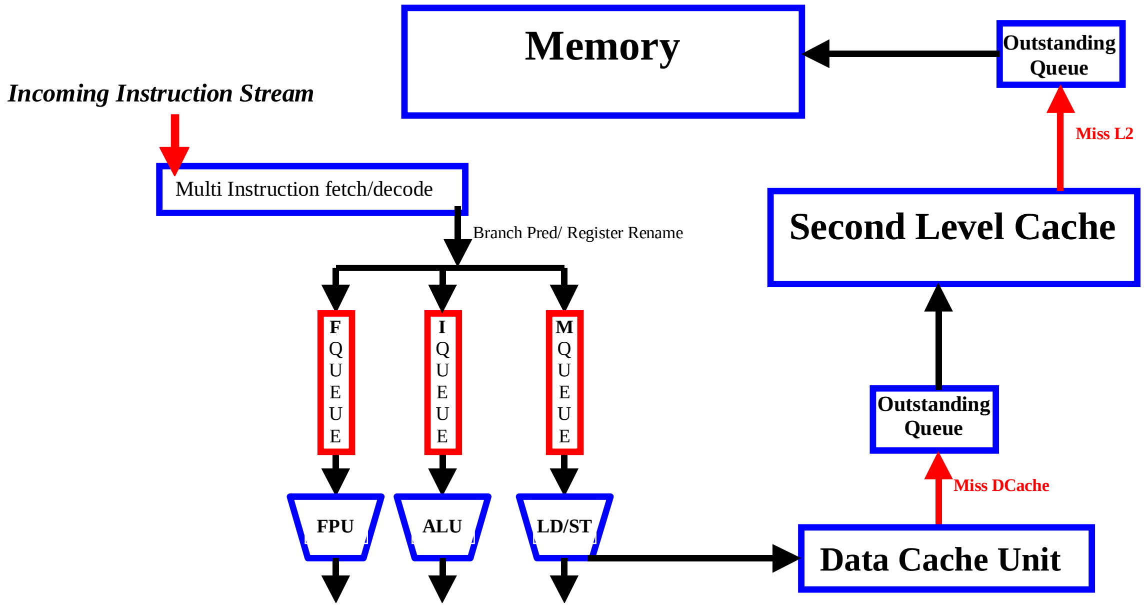

Figure 2 General pipeline model for CPU incorporating memory influence on stalls

In order to analyze the behavior of those queues illustrated in Figure 1, we need to measure the average inter-arrival distance in number of instructions, instead of cycles, which are dependent on both architecture and application. When we characterize an application, one of the keys is to separate the architectural factors so that a true workload characterization can be presented. This "number of instructions between two consecutive operations" idea is borrowed from the concept of *run-length* defined in [2].

$$\lambda_f = \frac{\text{\# graduated instructions}}{\text{\# graduated FP instructions}}$$

$$\lambda_i = \frac{\text{\# graduated instructions}}{\text{\# graduated INT instructions}}$$

$$\lambda_m = \frac{\text{\# graduated instructions}}{\text{\# graduated memory instructions}}$$

$$\lambda_{L1} = \frac{\text{\# graduated instructions}}{\text{\# L1 cache misses}}$$

$$\lambda_{L2} = \frac{\text{\# graduated instructions}}{\text{\# L2 cache misses}}$$

We define the preceding average terms for those queues illustrated in Figure 1. This λ value is a factor without a unit such that $1/\lambda_x$ is the probability of occurrence of instruction x over the incoming instruction stream. λ_{L1} and λ_{L2} refer to the occurrence of L1 and L2 misses. These are inclusive and a subset of overall memory instructions. To discount the effect of branch misprediction and the overhead impact of branch instructions, we also need to obtain the ratios of branch instructions and branch mispredictions to ensure the applications can be simplified as three major instruction flows (FP, Int, and Memory). On the other hand, the instruction cache miss ratio is also considered to see if the instruction fetch effect can be significant. The key of this methodology is to estimate which of the above λs can cause the stall of microprocessor due to the limitation of architectural or memory constraints.

We have devised a three-step process for analysis of the inner workings of a given microprocessor, based on our characterization. Our first step involves assumption of an infinite L1 cache to allow a focused study on CPI_0 modeling a generic system similar to Figure 1. To ease discussion, let us define G as the growth rate of queued instructions within the microprocessor. We must take into account the rate at which instructions graduate as well as the rate at which they are decoded giving:

$$G_x = \frac{\beta}{\lambda_x} - \Delta_x$$

where G_x is the growth rate for the x-queue of interest, β is the ideal instruction dispatching rate for the given microprocessor, λ_x is the measured distance between instructions of type x for a given code, Δ_x is

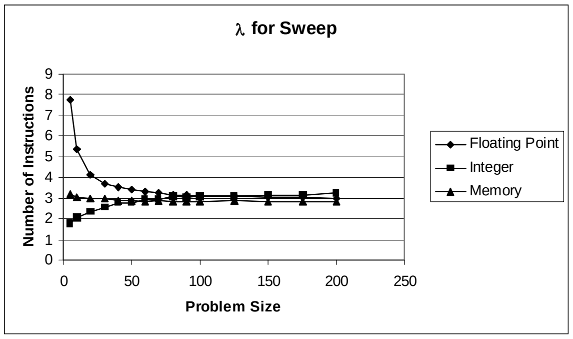

Figure 3 λ values for Sweep on Origin 2000

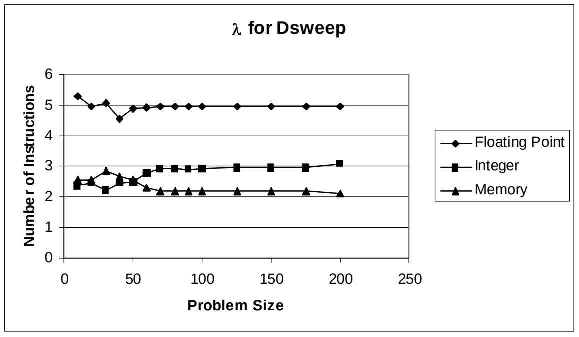

Figure 4 λ values for Dsweep on Origin 2000

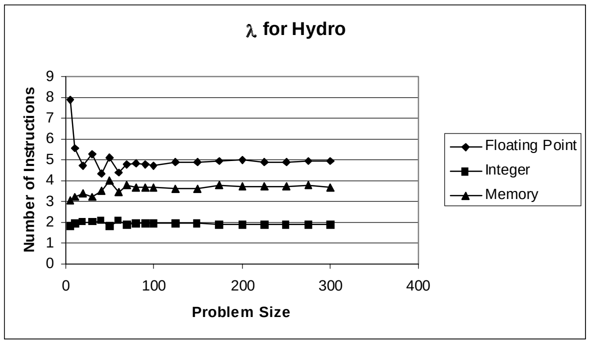

Figure 5 λ values for Hydro on Origin 2000

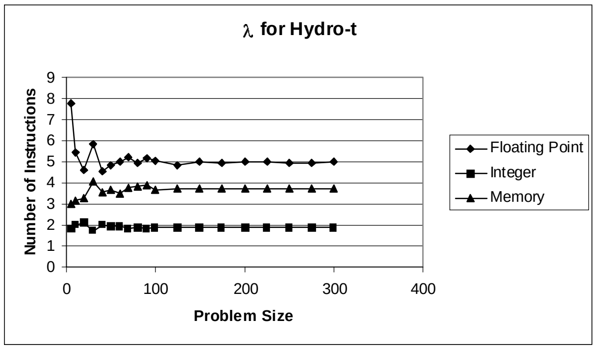

Figure 6 λ values for Hydro-t on Origin 2000

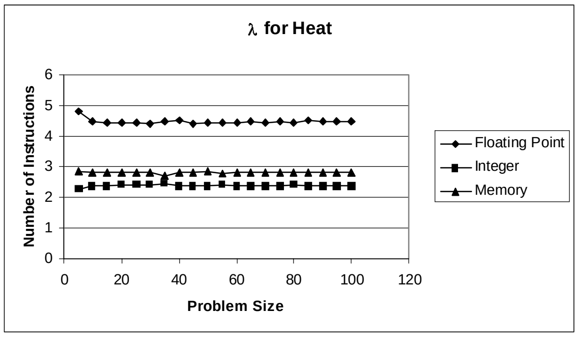

Figure 7 λ values for Heat on Origin 2000

the preset hardware graduation rate of the x-queue, and x is the current instruction type of interest, namely m, i, or f for memory, integer, or floating point instructions. We are interested in positive growth rates ($G_x>0$) for each queue in question. This formula, along with our infinite L1 cache assumption, allows us to approach a lower bound for the widely discussed CPI_0. Figures 3-7 show the steady state achieved for these λs. As a steady state is reached, positive growth rates will contribute to cpu stalls as any queue within the microprocessor reaches its capacity. These cpu stalls directly contribute to the underlying CPI_0. Multiple positive growth rates lead to contemplation of K, a threshold of maximum instructions in flight; in other words in some cases we must consider queue interaction as well as individual contributions to stalling. Since we assume an infinite L1 cache, indicate no branching effect, and ignore data dependency, calculations of CPI_0 based on λ values must give a lower bound to CPI_0. Current data supports these conclusions as work towards better approximations of CPI_0 continues.

In the second step, we focus on the first level of memory hierarchy taking into account the diagram in Figure 2. We no longer assume an infinite L1 cache and focus on architectural features that allow computational overlap at the queueing level. Most of

Microprocessor modeling:

…must assume infinite L1 cache
…models stalls within microprocessor due to on-chip architectural limitations
…must include overall assumptions

Memory modeling:

…do not assume infinite L1 cache
…models stalls within processor due to on-chip architectural limitations or the effect of memory hierarchy
…must include overall assumptions

Figure 9 Microprocessor vs. Memory modeling using instruction-level characterization

Necessary assumptions for modeling:

uniform distribution of instruction stream
convergence of each λ value in steady state
branch influence negligible
Icache effect negligible
data dependencies not considered
$1/\lambda_m + 1/\lambda_i + 1/\lambda_f = 1$

Figure 8 Overall modeling assumptions

today's superscalar microprocessors allow overlap of computation through support for outstanding cache misses. Through comparison of λ values when misses to L1 cache occur, we can infer the advantages of lengthening the number of outstandings supported on chip. This analysis is extendible to multi-layered caches and is not limited to this simple example. We define a term Q'_o as the maximum number of outstanding cache misses utilized by a code on a particular architecture.

This parameter gives us insight to the exploitation of outstanding misses for a particular code on a given architecture.

$$Q'_o = \frac{Q_m * \lambda_m}{\lambda_{L1}}$$

The third step of utilizing the instruction-level characterization attempts to draw conclusions related to cache size. When cache sizes across machines differ, λ values will reflect the performance gained. Larger cache should insinuate greater values for the respective λs. If not, then there is no significant performance gain attributable to a larger cache for a particular code.

This multi-level procedure based on real-time code measurements can provide both analysis of current performance and evaluation of possible gains/losses of simple architectural changes such as increasing queue length, increasing number of outstandings, or increasing cache size. A later section provides results obtained using this methodology on the test-bed discussed earlier.

We should note that we have given several provisos above, namely during different phases of the three step process. As shown in Figures 8 and 9, there are several underlying assumptions for each layer of analysis using the described characterizations. These provide a simplified modeling environment at the expense of applicability among various types of codes, but we feel the evolving techniques are useful and adaptable as advances are made to incorporate branch prediction and dependency modeling. Figures 8 and 9 show the overall assumptions, and the assumptions associated with each level of analysis.

Application Description

Three applications (5 codes), which form the building blocks for many nuclear physics simulations in Los Alamos National Laboratory, were used in this study. SWEEP is a three dimensional solver for the time independent, neutral particle transport equation on an orthogonal mesh [3]. The specific version used in these tests was a scalar-optimized "line-sweep" version [3] that involves separately nested, quadrant, angle, and spatial-dimension loops. In contrast with vectorized plane-sweep versions of SWEEP, there are no gather/scatter operations and memory traffic is

Table 1 Branch and icache characteristics for measured codes

	Branch Ratio (branch per instruction)	Miss Prediction Ratio (miss_pred per branch)	Branch Miss Ratio (miss_pred per instruction)	Icache Miss Ratio (icache_miss per instruction)
SWEEP	**0.0653**	**0.1365**	**0.0089**	**0.0002**
DSWEEP	**0.0570**	**0.0340**	**0.0017**	**0.0001**
HEAT	**0.0554**	**0.0393**	**0.0022**	**0.0017**
HYDRO	**0.1052**	**0.0988**	**0.0104**	**0.0088**
HYDROT	**0.1057**	**0.1126**	**0.0103**	**0.0087**

significantly reduced through "scalarization" of some array quantities. Because of these features, L1 cache reuse on SWEEP is fairly high (the hit rate is about 85%). A problem size of N implies N^3 grid points. Another version of SWEEP algorithm, DSWEEP is used in our experiments too. This is a vectorizable implementation of the diagonal line-sweep.

HYDRO is a two-dimensional explicit Lagrangian hydrodynamics code based on an algorithm by W. D. Schulz [4]. HYDRO is representative of a large class of codes in use at the Laboratory. The code is 100% vectorizable. An important characteristic of the code is that most arrays are accessed with a stride equal to the length of one dimension of the grid. HYDRO-T is a version of HYDRO in which most of the arrays have been transposed so that access is now largely unit-stride. A problem size of N implies N^2 grid points.

HEAT solves the implicit diffusion PDE using a conjugate gradient solver for a single timestep. The code was written originally for the CRAY T3D using SHMEM. The key aspect of HEAT is that its grid structure and data access methods are designed to support one type of adaptive mesh refinement (AMR) mechanism, although the benchmark code as supplied does not currently handle anything other than a single-level AMR grid (i.e. the coarse, regular level-1 grid only). A problem size of N implies N^3 grid points.

All of these benchmark codes are run on two MIPS R10000-based systems: the SGI PowerChallenge and the Origin2000. Table 1 exhibits branch ratios, branch misprediction ratios, and the instruction cache miss ratios for all these codes. It is clear from Table 1 data that both branch and instruction cache effect can be negligible. Under this condition, the performance study of these codes can focus on the impact of the three major instruction flows (FP, Int, & Memory).

Figures 3-7 show the variations of the λs for all 5 benchmark codes in this study. These figures demonstrate that the λs converge to constant values with increasing problem sizes. This is understood as the instruction flow pattern of a problem that reaches its steady state. This phenomenon proves that λs can be used in characterizing these codes once they reach the steady state.

Performance Bottleneck Estimation

This new instruction-level workload characterization technique is first applied to two R10000-based systems to estimate the application performance bottleneck. This SGI microprocessor is a state-of the art 4-way superscalar microprocessor supporting out-of-order and speculative execution as well as non-blocking cache and register renaming. The R10000 microprocessor has the following major architecture constraints to cause a CPU stall [6] (besides branch and instruction cache effects): a). one of the three main queues is full; b). outstanding miss queue is full; c). the number of total instructions in all three queues reaches its maximum 32; d). all renaming registers are consumed; e). there is more than one back-to-back write-back from L1. On the R10000 processor, both the F queue and the M queue have 16 entries each. The I queue can accommodate 16 instructions. As a good first-order approximation [7], at each cycle, the load/store unit can execute one memory instruction. The two ALUs can graduate up to two integer instructions per cycle and the FPUs complete up to two floating-point instructions each cycle. The total number of instructions in flight on a R10000 is 32. The outstanding miss queue length is less than 2 on the PowerChallenge and 4 on the Origin2000. According to [6], since a R10000 has 64 registers for renaming, it is unlikely that all 64 registers are exhausted before any other limit is reached. Also, the limit e) of L1 write-back buffer may not be reached most of the time. Therefore, we can focus on the other 3 constraints.

Utilizing these instruction-level characteristics, we calculate the growth rates for each code over both machines in Table 2. Due to their architectural

Table 2 Measured growth rates

	Sweep			Dsweep			Heat			Hydro			Hydro-t		
	G_m	G_i	G_f	G_m	G_i	G_f	G_m	G_i	G_f	G_m	G_i	G_f	G_m	G_i	G_f
PowerChallenge	**0.41**	**-0.73**	**-0.68**	**0.84**	**-0.65**	**-1.19**	**0.43**	**-0.32**	**-1.11**	**0.08**	**0.11**	**-1.19**	**0.08**	**0.12**	**-1.20**
Origin 2000	**0.42**	**-0.76**	**-0.66**	**0.89**	**-0.70**	**-1.19**	**0.42**	**-0.32**	**-1.11**	**0.09**	**0.10**	**-1.19**	**0.08**	**0.12**	**-1.20**

Table 3 Cache miss distances

	Sweep		Dsweep		Heat		Hydro		Hydro-t	
	λ_{L1}	λ_{L2}	λ_{L1}	λ_{L2}	λ_{L1}	λ_{L2}	λ_{L1}	λ_{L2}	λ_{L1}	λ_{L2}
PowerChallenge	**26.6**	**112.8**	**12.5**	**34.6**	**15.5**	**62.2**	**13.5**	**78.8**	**30.3**	**274.2**
Origin 2000	**24.9**	**122.9**	**12.7**	**38.0**	**15.5**	**62.6**	**13.4**	**219.4**	**30.3**	**290.1**

Table 4 Outstanding miss utilization, Q'_o

	Sweep	Dsweep	Heat	Hydro	Hydro-t
PowerChallenge	**1.7**	**2.8**	**2.9**	**4.4**	**2.0**
Origin 2000	**1.8**	**2.7**	**2.9**	**4.4**	**2.0**

similarity, the growth rates are identical across PowerChallenge and Origin 2000. For Sweep, Dsweep, and Heat the only positive growth rate is given in G_m. This leads us to declare the memory instruction growth rate as our *limiting factor* for these codes on these machines. A *limiting factor* is the key contributor to stalls within the microprocessor (excluding dependencies and memory latency as we assume infinite L1 cache). For these codes, it is very likely the memory queue will fill, leading to stalls in decoding as entries graduate slower than they arrive. For Hydro and Hydro-t, we have positive growth rates for the memory and integer queues. This leaves us two possibilities for the *limiting factor*. The queue associated with the maximum of the two growth rates in the ideal case would fill first, namely the integer queue. This can only happen however, if the maximum instruction threshold K is not reached. As mentioned above, K=32 for the MIPS R10000. Since the memory and integer queue lengths are both 16, we cannot reach the maximum number of instructions prior to stalling on a single queue. Thus, the limiting factor for both of these codes will be the integer instruction growth rate.

For the second step of the process, we no longer assume infinite L1 cache, and focus on the λs for the L1 cache misses. In Table 3, the values for λ_{L1} over the codes and machines are given. As discussed earlier, the PowerChallenge allows two outstanding misses. In this particular case, if the number of maximum outstandings utilized by a code is less than 2, the outstanding misses are not fully utilized. The results when calculating Q'_o are given in Table 4. Note the values are very close across machines. Q'_o is influenced by architecture (the number of memory queue entries) and code characteristics (λ values for memory and L1). In this case, even though the memory hierarchies differ, the queue entries are the same since they are on chip. Also, since the same compiled code is executed on both machines, the instruction streams are identical. Herein lies the usefulness of this technique in that it allows us to isolate this particular architectural feature, namely outstanding misses. From Table 4, we can see the maximum number of misses that could be utilized for a given code-machine combination. For PowerChallenge, code-machine combinations less than 2 under-utilize the existing architecture. This happens in the case of Sweep and Hydro-t. For Dsweep, Heat, and Hydro, on the PowerChallenge, the architectural specification of only 2 outstanding misses could certainly contribute to stalls within the microprocessor. For the Origin 2000, we have 4 outstanding misses. Sweep, Dsweep, Heat, and Hydro-t all under-utilize the available outstanding misses. In Hydro however, stalls could occur within the microprocessor due to filling the outstanding miss queue.

For the third part of the process, we observe the λ_{L2} values in Table 3. The frequency of L2 misses shows a sharp decline from the PowerChallenge to the Origin 2000 for Hydro. This indicates that Hydro is the only code that gains an advantage from the larger L2 cache (2MB L2 on the PowerChallenge, 4MB L2 on the Origin2000). This is also validated in the empirical memory model [1] and the statistic model [5].

Future Work

We intend to validate more thoroughly the proposed relationship of λ values to CPI_0 using simulators. We would also like to expand to more comprehensive equations involving the relationships discussed above. We are currently attempting to extend this characterization to commercial workloads as we feel such extension is complementary to the techniques discussed. Finally, memory bandwidth, branch/icache impact, and data dependency should be incorporated in an evolving model to extend the applicability and validity of this modeling technique.

References:

1. Lubeck, O.M, Luo, Y., and Wasserman, H.J. et al, An Empirical Hierarchical Memory Model Based on Hardware Performance Counters, PDPTA'98, Las Vegas, July 13-16, 1998.
2. Bianchini, R., Lim, B., *Evaluating the Performance of Multithreading and Prefetching in Multiprocessors*, Journal of Parallel and Distributed Computing, N.37, p83-97, 1996.
3. Koch, K. R., Beker, R. S., and Alcouffe, R.E., Solution of the First-Order Form of the 3-D Discrete Ordinates Equation on a Massively Parallel Processor, Trans. of the Amer. Nuc. Soc., 65, 198, 1992.
4. Schulz, W.D., *Two-Dimensional Lagrangian Hydrodynamic Difference Equations*, Methods in Computational Phys. Vol 3, p1, 1964.
5. Sun, X. H., Cameron, K. W., et al., A Hierarchical Statistic Methodology for Advanced Memory System Evaluation, submitted to IPPS'99, Sept. 1998.
6. Schwarzmeier, J. (SGI/Cray), *Private Communications*, Sept. 1997.
7. Turner, S. (SGI/Cray), *Private Communications*, Mar. 1998.